Efficient First Order Inductive Learner on Spark

Lab Report: Distributed Big Data Analytics, 2017

Georgiy Shurkhovetskyy, Eskender Haziiev

The presented report deals with realization of First Order Inductive Learner. The work is based on the approach by J.R. Quinlan and is developed using Spark framework. A prediction rule in a form of a set of Horn clauses for a target relation that consistent with given positive examples and not cover any given negative examples is displayed result. Each example is given as positive or negative tuple. As input data the application receives an information about the name and a set of positive and negative tuples of target predicate, a list of names and sets of positive tuples of predicates that can appear in a right-hand side of the rule. Three successful experiments were carried out.

1 Problem definition

1.1 Introduction

Learning theory considers any object as fixed collection of attributes and there is solved a problem on belonging of this object to one of a bounded number of mutually exclusive classes. Using inductive approach the above problem is clarified as follows. Suppose we have an information on belonging some training set of objects having the similar structure to one of two mutually exclusive classes. It is necessary to find a rule for predicting the class of an unseen object having the similar structure of a set of the attributes. Note that a set of object attributes does not have to coincide with the complete set of attributes and can only be subset of the last one.

Several methods are used to solve such problem. One of them (so-called a divide-and-conquer method) is based on a construction of a decision tree (DT) from a given training set, then if-then rules are extracted from DT. An algorithm consists of two operations: a selection of a test based on one attribute and a separation of the training set into subsets corresponding to one of the mutually exclusive results of this test. Then these operations are repeated to obtained subsets. The process is ended when all examples from each obtained subsets are belonged to a single class (i.e. labeled as a leaf). However, the construction of a compact DT strongly depends on the successful choice of the test, the algorithm is greedy and takes a lot of time [Quinlan, 1990].

To get rules from a training set an inductive method (so-called a covering method, AQ family) directly uses the concept of separate and conquers. This approach includes: a finding of a conjunction of conditions that is satisfied by some objects in the target class, but no objects from another class (in fact, this is a test); appending of obtained conjunction in disjunctive logical expression and remove all objects that satisfy it. The procedure repeats if there are still remaining objects of the target class. The "bottleneck" of this approach is based on its dependence on specific training examples [Quinlan, 1990].

Using concrete example (see network at Figure 1) Quinlan [1990] showed the principal weakness of described above methods and suggested new inductive approach realized in FOIL (First Order Inductive Learner). FOIL is a system which learns function-free Horn clause definitions from data expressed as relations in terms of itself and other relations. Quinlan [1990] presented the results on six problems addressed by other learning systems, see more in [Quinlan and Cameron-Jones, 1993, 1995]. Noted that

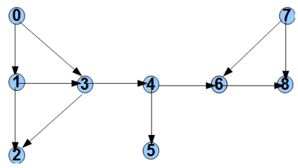

Figure 1: Illustrated example by Quinlan [1990].

Muggleton and De Raedt [1994] pointed out that FOIL system "relates to the greedy TDIDT-algorithms and the AQ family of algorithms...was the first widely known demonstration that the first order learning ... works efficiently on a broad range of problems involving large and noisy datasets".

Our report deals with the following problem: *using Spark framework to realize FOIL which finds a learning rule in form of a set of Horn clauses for the target relation that consistent with given positive examples and not cover any given negative examples.*

It should be noted that *the desired application is of interest for a Big Data problem* for the following reasons. The Big Data problems usually handles large amounts of input data in the learning process. The effectiveness of the process is directly related to the representation of the training data and a result. The FOIL as inductive approach allows to derive from a given training data a learning rule in the analytical form, namely as Horn clauses. A brief analytical form of the obtained learning rule allows us to obtain a prediction on a belonging to one or another mutually exclusive class for any unseen object quickly enough.

1.2 Learning Logical Definitions from Relations: concepts and algorithm

Based on [Quinlan, 1990, Quinlan and Cameron-Jones, 1993] we will outline the basic concepts and the main ideas of learning logical definitions from relations.

Most often an object may belong to one of two classes or we have to predict whether an object belongs to some class. To predict the learning rules represented by Horn clauses can be used. So we discuss a subject of present report namely for this case.

FOIL input usually consists of information on several relations. One of them is target relation and its Horn clauses representation is to be founded.

Preliminarily let us give some basic definitions.

Definition 1. A literal L may be unnegated predicate P or negated predicate $\neg P$.

Definition 2. A claus body is a conjunction of literals.

Definition 3. A clause head is predicate.

Definition 4. A Horn clause consists of a clause head and a clause body in a form

$$P \leftarrow L_1, L_2, ..., L_n \quad , \tag{1}$$

where each L_i, $i = 1, 2, \ldots, n$, is literal.

Definition 5. A learning rule for predicate P is a collection of Horn clauses each with head P.

Note that Definition 1 and Definition 4 are important for recursive definition of predicate (it is seen when in (1) we use P in clause body instead some L_i).

The predicates can be defined *extensionally* as a list of tuples for which the predicate is true, or *intensionally* as a set of Horn clauses (using them we can compute whether the predicate is true).

Definition 6. k-tuple is a finite sequence of k constants, denoted by $< a_1, a_2, \ldots, a_k >$.

Below we also will use a short term *tuple* instead a term $k - tuple$.

Definition 7. A tuple satisfies a rule if it satisfies one of the Horn clauses of this rule.

For every relation a set of the tuples which belong to this relation (denoted by $\oplus$) is given. So such set is given for target relation too.

For example represented at Figure 1 the relation $linked - to(X, Y)$ may be given:

$$\begin{split} linked - to(X, Y) = \oplus\{< 0, 1 >, < 0, 3 >, < 1, 2 >, < 3, 2 >, < 3, 4 >, \\ < 4, 5 >, < 4, 6 >, < 6, 8 >, < 7, 6 >, < 7, 8 >\}. \end{split} \tag{2}$$

Each tuple $< X, Y >$ in (2) means that node X directly linked (connected) to node Y.

For a target relation a set of $\oplus$ tuples is also given. A set of the tuples not belonging (denoted by $\ominus$) to the target relation may be given. The statement is introduced: if some tuple is not included in set $\oplus$ tuples then it is $\ominus$ tuple.

Let a target relation be $can - reach(X, Y)$ which means that exists or not exists a path from node X to node Y. Then we have for above example:

$$
\begin{aligned}
can-reach(X,Y) = \ominus\{ &< 0,0 >, < 0,7 >, \\
&< 1,0 >, < 1,1 >, < 1,3 >, < 1,4 >, < 1,5 >, < 1,6 >, < 1,7 >, < 1,8 >, < 2,0 >, \\
&< 2,1 >, < 2,2 >, < 2,3 >, < 2,4 >, < 2,5 >, < 2,6 >, < 2,7 >, < 2,8 >, < 3,0 >, \\
&< 3,1 >, < 3,3 >, < 3,7 >, < 4,0 >, < 4,1 >, < 4,2 >, < 4,3 >, < 4,4 >, < 4,7 >, \\
&< 5,0 >, < 5,1 >, < 5,2 >, < 5,3 >, < 5,4 >, < 5,5 >, < 5,6 >, < 5,7 >, < 5,8 >, \\
&< 6,0 >, < 6,1 >, < 6,2 >, < 6,3 >, < 6,4 >, < 6,5 >, < 6,6 >, < 6,7 >, < 7,0 >, \\
&< 7,1 >, < 7,2 >, < 7,3 >, < 7,4 >, < 7,5 >, < 7,7 >, < 8,0 >, < 8,1 >, < 8,2 >, \\
&< 8,3 >, < 8,4 >, < 8,5 >, < 8,6 >, < 8,7 >, < 8,8 >, \}. \quad (3)
\end{aligned}
$$

$$
\begin{aligned}
can-reach(X,Y) = \oplus\{ &< 0,1 >, < 0,2 >, < 0,3 >, < 0,4 >, < 0,5 >, < 0,6 >, \\
&< 0,8 >, < 1,2 >, < 3,2 >, < 3,4 >, < 3,5 >, < 3,6 >, < 3,8 >, \\
&< 4,5 >, < 4,6 >, < 4,8 >, < 6,8 >, < 7,6 >, < 7,8 >\}. \quad (4)
\end{aligned}
$$

The problem definition: To find a learning rule as a set of Horn clauses for a target relation that consistent with all given $\oplus$ tuples and not cover any given $\ominus$ tuples.

2 Approach

In this project we follow the algorithm FOIL described above and presented in Table 1. It is not intended to apply any parallelization of the process. For implementation the Spark framework and Scala programming language should be used. It is intended to use training data based on [Horvath, 2016] and [Quinlan, 1990].

Pazzani and Kibler [1992] formulated two designs of the FOIL algorithm, one of them is presented below in Table 1.

Table 1. FOIL Design II (by Pazzani and Kibler [1992]).

1	Let Pred be the predicate to be learned
2	Let Pos be the positive examples
3	Until Pos is empty do:
4	Let Neg be the negative examples
5	Set Body to empty
6	Let Old be those variables used in Pred
7	CallLearnClauseBody
8	Add Pred $\leftarrow$ Body to the rule
9	Remove from Pos all examples that satisfy the Body
7a	Procedure CallLearnClauseBody
7b	Until Neg is empty do:
7c	For each predicate-name P
7d	For each variabilization L of P
7e	Compute information gain of L and its negation
7f	Select literal L with most information gain
7g	Conjoin L to Body
7h	Add any new variables to Old
7i	Let Pos be all extensions of Pos that are satisfied by the literal
7j	Let Neg be all extensions of Neg that are satisfied by the literal

FOIL has two levels of a processing: an *outermost level* and a *inner level.*

At the outermost level FOIL starts with a training set containing all $\oplus$ and $\ominus$ tuples, constructs a function-free Horn clause which consistent with some $\oplus$ tuples, removed the covered $\oplus$ tuples from the training set and continues with the search for next clause. This operation is ended when the set of $\oplus$ tuples becomes empty.

At the inner level FOIL constructs current Horn clause in form (1). The current clause is expands by adding new literals to the body. In general inner loop starts with assignment current training set T_1 to training set and initialization $i = 1$; then while T_i contains $\ominus$ tuples a new literal L_i is found and is added to the body of current clause, new current training set T_{i+1} is generated based only on tuples from T_i which are satisfied to literal L_i, variable i is incremented and inner loop is repeated. The inner loop is ended when set of $\ominus$ tuples in T_i becomes empty.

The operations 7c–7f in CallLearnClauseBody select one of the possible literals. The having greatest positive gain literal is added to a current clause body. At each i-nd iteration of the inner loop the literal L_i is looking for one of the possible literals:

$$gain(L_i) = n_i^{\oplus\oplus} \cdot (I(T_i) - I(T_{i+1})), \quad I(T_i) = -\log_2(n_i^{\oplus}/(n_i^{\oplus} + n_i^{\ominus})), \tag{5}$$

where $n_i^{\oplus}$ — a number of $\oplus$ tuples in T_i, $n_i^{\ominus}$ — a number of $\ominus$ tuples in T_i, $n_i^{\oplus\oplus}$ — a number of the $\oplus$ tuples in T_i which are represented by one or more tuples in T_{i+1}.

The construction of new training set is also nontrivial, see 7h-7j. New training set consists of the $\oplus$ and $\ominus$ tuples which may have a larger length if a new variable was appeared in the clause body. The following operations are realized: for each tuple t in T_i and for each binding b of any new variables introduced by the literal L_i do: if the tuple $< t, b >$ satisfies L_i then add $< t, b >$ to T_{i+1} with the same label ($\oplus$ or $\ominus$) as t.

3 Implementation

For the implementation of described above algorithm Scala programming language and Spark framework were used. The predicates were generated as follows: $predicate(tuples)$, where $predicate$ is predicate name, $tuples$ is a list of the tuples for this predicate. The background knowledge is a list of $predicate(tuples)$ characterizing positive examples of predicates which can appear as literals on the right-hand side of the Horn clause.

To generate a learning rule we use the basic methods of our own class $KnowledgeBase$:

1. $load$ which loads data from input stream and store the sets of positive examples and negative examples of target predicate, background knowledge $\mathcal{B}$, as $DataSource$ objects within class fields (in fact $load$ enters a training data);

2. $foil$ runs FOIL algorithm which uses the sets of positive examples and negative examples and background knowledge and obtains the rules.

Describing the work of the method $foil$ it should be noted that the sequence of operations is performed:

2.1. $generateCandidates$ generates list of $candidates$ with all possible combinations of variables from background knowledge predicates;

2.2. for each $PositiveExample$ until $PositiveExamples \neq \emptyset$ a set $NegativeExamples$ is taken;

2.2.1. for each $NegativeExample$ until $NegativeExamples \neq \emptyset$ every candidate from $candidates$ is taken;

2.2.2. $findMaximalClause$ finds "maximal" Horn clause, i.e. finds a predicate having maximal weighted gain calculated by formula (5);

2.2.3. a clause body is updated by concatenation of the predicate having maximal weighted gain.

2.3. the generated rule is displayed.

Now consider method $findMaximalClause$ having as Input $targetPredicate$ (i.e. left-hand side of the rule), $bodyPredicates$ (i.e. already generated right-hand side of the rule), the sets $PositiveExamples$ and $NegativeExamples$. As mentioned above this method has as Output a predicate having maximal weighted gain calculated by (5). $findMaximalClause$ executes the call $foilAlgorithm$; and returns set of the tuples covered by the rule and weighted gain, obtained from n^+, n^- and n^{++}.

The method $foilAlgorithm$ is designed for check tuples in FOIL algorithm. As Input it has $targetTuples$ to be checked, $targetPredicate$ in left-hand side of the rule, $bodyPredicates$ already generated right-hand side of the rule. As Output $foilAlgorithm$ obtains Scala tuple containing calculated n^+, n^- and n^{++}, and the set of the tuples covered by the rule.

4 Evaluations

The efficiency of our application is estimated by three indicators. The first and second ones are connected with the evaluation of the correctness of the obtained result, i.e. whether the obtained rile is consistent with the given training data. Here we analyzed a proportion of positive examples and a proportion of negative examples that are covered by the obtained rule (in percentage terms). It is obvious that these indicators satisfy metrics properties and the application can be evaluated as "efficient" if it has a high first rate and a low second one. The third indicator characterizes the speed of data processing. Here we analyzed a running time of program (in seconds). The application can be evaluated as "efficient" if it has a low running time.

Three experiments were running in Spark framework and the learning rules in form (1) were obtained. In the first experiment we used training data from [Horvath, 2016]. The target relation is 2-ary predicate $daughter(X_1, X_2)$; 2-ary predicate $parent(X_1, X_2)$ and 1-ary predicate $female(X)$ can be used in learned clause body. The predicates names are replaced by initial letters d, p and f correspondingly. The training data is following:

$$\mathcal{B} = \{p(tom, bill), p(tom, emily), p(ann, tom), p(ann, mary), f(emily), f(mary), f(ann)\},$$
$$PositiveExamples = \{d(emily, tom), d(mary, ann)\},$$
$$NegativeExamples = \{d(emily, ann), d(tom, ann)\}. \quad (6)$$

The method $generateCandidates$ uses the literal candidates:

$$candidates = \{p(X_2, X_1), p(X_1, X_2), p(X_1, Y_1), p(Y_1, X_1), p(X_2, Y_1), p(Y_1, X_2),$$
$$p(X_1, X_1), p(X_2, X_2), f(X_1), f(X_2)\}. \quad (7)$$

We obtained the resulting rule $d(X_1, X_2) \leftarrow f(X_1)\ p(X_2, X_1)$. Indeed, the proposition (X_1 is a daughter of X_2) is a consequent of conjunction of antecedents (X_1 is a female) and (X_2 is the parent of X_1). In fact, Mary is Ann's daughter and Eve is Tom's daughter; for example, Eve is not a daughter of Ann, see Figure 2(a).

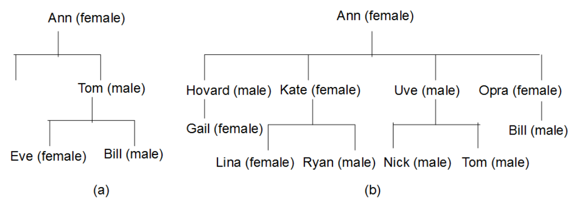

Figure 2: Two *ancestor-descendant* trees.

In the second experiment to get the rule for the same target relation we used a more complex tree $ancestor - descendant$, see Figure 2(b). The training data is following:

$$\mathcal{B} = \{p(o,b), p(u,t), p(u,r), p(k,n), p(k,l), p(h,g), p(a,o), p(a,u), p(a,k), p(a,h), f(a),$$
$$f(g), f(l), f(o), f(k)\},\ NegativeExamples = \{d(b,a), d(b,o), d(t,u), d(t,a), d(r,u),$$
$$d(n,k), d(n,a), d(r,a), d(l,a), d(g,a), d(u,a), d(h,a)\},$$
$$PositiveExamples = \{d(o,a), d(k,a), d(l,k), d(g,h)\}, \quad (8)$$

the person names also are denoted by initial letters and the same literal candidates are used. The obtained rule is the same above. Indeed, Kate and Opra are Ann's daughters, Lina is Kate's daughter and Gail is Hovard's daughter; at the same time, for example, Tom is not a daughter of Kate or Ann, Gail is not a daughter of Ann, see Figure 2(b).

The goal of the third experiment was to build a rule for a target relation $son(X_1, X_2)$ based on training data shown at Figure 2(b) and we had to add a predicate $male(X)$ (denoted by letters s and m). Here instead (6) and (7) we used

$$\begin{gathered}\mathcal{B} = \{p(o,b), p(u,t), p(u,r), p(k,n), p(k,l), p(h,g), p(a,o), p(a,u), p(a,k), p(a,h), m(b),\\ m(u), m(t), m(r), m(n), m(h)\}, \quad NegativeExamples = \{s(b,a), s(t,a), s(r,a), s(n,a),\\ s(l,a), s(g,a), s(o,a), s(k,a), s(l,k), s(g,h))\},\\ PositiveExamples = \{s(u,a), s(b,o), s(t,u), s(r,u), s(n,k), s(h,a)\},\\ candidates = \{p(X_2,X_1), p(X_1,X_2), p(X_1,Y_1), p(Y_1,X_1), p(X_2,Y_1), p(Y_1,X_2),\\ p(X_1,X_1), p(X_2,X_2), m(X_1), m(X_2)\}. \quad (9)\end{gathered}$$

The obtained rule is the same above: $s(X_1, X_2) \leftarrow m(X_1)\ p(X_2, X_1)$. Indeed, Hovard and Uve are Ann's sons, Nick and Tom are Uve's sons, Ryan is son of Kate and Bill is son of Opra. However, for example, Tom is not a son of Kate or Ann, Ryan is not a son of Opra, Lina is not a son of Opra or Hovard, see Figure 2(b).

Table 2. Efficiency indicators.

No Experiment	Proportion of covered $\oplus$(%) examples	Proportion of covered $\ominus$ examples (%)	Running time (sec)
1	100	0	0.188
2	100	0	0.207
3	100	0	0.200

All computations were performed on the computer with the following specifications: Processor — Intel(R) Core(TM) i5-7200; CPU 2.50GHz-2.70GHz; Installed memory (RAM) = 8GB; System type — 64-bit Operating System Windows 10 Home.

5 Project timeline

Author list:

1. Georgiy Shurkhovetskyy — Team leader; 2. Eskender Haziiev — Team member.

t

Week	Planned work	Implementer
29.05.2017-11.06.2017	Study of literature, writing project timeline Study of literature, writing References	Team leader Team member
11.06.2017-18.06.2017	Building of algorithm structure Progress presentation construction	Team leader Team member
19.06.2017-25.06.2017	Building of data structure Development of classes	Team leader Team member
26.06.2017-02.07.2017	Choice of effectiveness evaluation technique Debugging the program	Team leader Team member
03.07.2017-09.07.2017	Debugging the program	Team leader Team member
10.07.2017-16.07.2017	Test 1 and evaluation of its results Test 2,3 and evaluation of its results	Team leader Team member
17.07.2017-23.07.2017	Writing 1,2,5 sections of the report Writing 3,4 sections of the report, Final presentation construction	Team leader Team member Team member

References

T. Horvath. Learning in logic V. In *Learning from Non-Standard Data, WS 2016/17*. University of Bonn, Fraunhofer IAIS, Sankt Augustin, Germany, 2016.

S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19(20):629–679, 1994.

M. Pazzani and D. Kibler. The utility of knowledge in inductive learning. *Machine Learning*, 9(1):57–94, 1992.

J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, (5):239–266, 1990.

J.R. Quinlan and R.M. Cameron-Jones. Foil: A midterm report. In *Machine Learning: ECML-93*, pages 3–20. Springer-Verlag, Berlin, Heidelberg, 1993.

J.R. Quinlan and R.M. Cameron-Jones. Induction of logic programs: Foil and related systems. *New Generation Computing*, 13(3):287–312, 1995.